# Serialization and Applications for the Gen Probabilistic Programming Language

by

Ian Limarta

S.B. in Electrical Engineering and Computer Science
Massachusetts Institute of Technology (2022)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2023

| | |
|---|---|
| Authored by: | Ian Limarta<br>Department of Electrical Engineering and Computer Science<br>August 11, 2023 |
| Certified by: | Vikash Mansinghka<br>Principal Research Scientist<br>Thesis Supervisor |
| Accepted by: | Katrina LaCurts<br>Chair, Master of Engineering Thesis Committee |

# Serialization and Applications for the Gen Probabilistic Programming Language

by
Ian Limarta

Submitted to the Department of Electrical Engineering and Computer Science
on August 11, 2023, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

## Abstract

Probabilistic programming has emerged as a powerful framework for building expressive models that can handle uncertainty in a wide range of applications. Serialization, the process of converting data structures or objects into a format suitable for storage or transmission, plays a crucial role in the development and execution of probabilistic programs. Efficient serialization techniques are essential for tasks such as data persistence, distributed computation, and data exchange between different programs or machines. We delve into specific challenges and considerations unique to probabilistic programming for serialization. Probabilistic models often involve complex structures, including nested random variables, hierarchical dependencies, and potentially infinite or unbounded dimensions. Serializing samples from such models requires careful handling of these complexities, including strategies for preserving model fidelity, dealing with modeling dependencies, and specializing for disk representations. In this thesis, we discuss twofold objectives for the Gen probabilistic programming model. The first establishes a formalism for serializing (and deserializing) traces as an interface that respects the existing Gen interfaces and faithfully reconstructs data objects from disk. We highlight challenges for efficient serialization for Gen's DSLs. The second objective is to show how serialization routines common in other areas of computing transfer well to Gen. We show how serialization provides easiers means for visualizations, remote computing, and training inference approximators.

Thesis Supervisor: Vikash Mansinghka
Title: Principal Research Scientist

## Acknowledgments

This project would not have been possible without the gracious help of the members at the Probabilistic Computing Project. I would like to thank my supervisor Vikash Mansinghka for his generosity and providing me the opportunity to spend the year learning about probabilistc computing. It was a time of fruitful discovery and alignment of my interests. My time at ProbComp would not have been the same had McCoy Becker not ramped the thesis work to fourth gear. I thank him for all his help and advice as well as admire his unusually high degree of patience. I'd also like to thank Nishad Gothoskar and Matin Ghavamizadeh for being absolute OGs with debugging and theory help. I extend my warmest thanks for Alex Lew who clarified Gen's internals and graciously reviewed the writing. I would also like to thank Arijit Dasgupta, Tan Zhi-Xuan, Andrew Bolton, Cameron Freer, and Austin Garrett for helpful discussions along the way. The lab would be a circus if it wasn't for Amanda and Rachel's incredible support. Again thanks. I would also like to acknowledge the incredible work McCoy and Matin have developed which this thesis stands on.

Finally, I thank my parents whose unconditional love brought me to where I now am. I am forever thankful for them.

This research was funded by the DARPA Machine Common Sense program, under (sub)contract CW3013540.

# Contents

# List of Figures

# List of Tables

# Chapter 1
# Introduction

Probabilistic programming language (PPL) shave become an increasingly popular paradigm for modeling using Bayesian inference. At its core, a PPL automates Bayesian inference procedures and lets the user focus on the model specification instead. By abstracting away inference, these languages can greatly simplify the complexity and redundancy of writing algorithms by hand. Today PPLs have become increasingly sophisticated, and several languages support nearly abitrary execution flows such as looping and branching. Under the hood, a PPL may produce data objects that keep track of state conducive for efficient inference. Objects emitted may be structured according to model specifications, the types of inference workloads, and runtime environment. One common data object found in several PPLs is a container storing stochastic events with associated probabilities generated from a model. Gen, a PPL designed for custom inference, heavily uses this concept in the form of *traces* to sample from models, condition distributions, and score generated samples. Thus, traces are crucial data objects emitted from implementations of generative models and are of interest for many applications in Gen. Often users use the random choices in the trace to gain insight into the generative model or use them as a debugging information to diagnose inference convergence. For computationally expensive generative models, however, maintaining traces in memory is less ideal and is not a solution for long-term storage. Without a tool to save traces to disk, users would need to re-run the model everytime to recollect data. This becomes cumbersome for recording experiemental results, sharing data between machines, and reliably resuming inference after an indefinite amount of time.

Frequently in many languages, users serialize objects to disk using well-supported tools. Different serializers write objects in various formats (e.g. binary or BSON) and are geared to different types of data or read workloads. However, most serializers fail to save out ephemeral data created during the runtime of the program such as pointers or generated code, and implementations of Gen's traces heavily use them. To overcome these difficulties, this thesis outlines a serialization specification for Gen that handles trace objects while respecting Gen's existing specification. We show how these interfaces remain composable with the modeling interfaces and explain efficient implementations for several of the modeling constructs.

The benefits of serialization come in many flavors. First this enables typical serialization routines prevalent in many data science applications such as visualization and storage. In the PPL setting, serialization tools also provide a convenient way of performing inference between different runtimes, transmitting traces between machines, and running memory-intensive inference. The stochastic nature of generative models means that a reliable storage format ensures replication across different executions and machines. Moreover, serialization can be used in novel ways for amortizing training cost for parameterized models. For example, finding an approximate distribution for an expensive generative process against some objective can be slow if the model needs to be sampled many times. Serializing out samples from the model and reusing these samples can help speed up the training process.

## 1.1 Contributions

This thesis makes the following contributions:

1. A formal description of serialization and deserialization interfaces for the Gen programming language. We demonstrate how these interfaces remain composable and demonstrate how to implement these interfaces efficiently for Gen's DSLs.

2. Example applications of serialization. We show how common uses of serialization can be extended to probabilistic programs. Creating visualizations, out-of-core computation, and interprocess communication across machines often use serialization as a means to accomplish tasks. We also show how to save out samples from slow generative processes to train inference approximators.

Chapter 2 provides background on Gen's design and specification. We review sequential importance resampling and provide a overview of amortized EUBO training in the setting of variational inference. Chapter 3 presents the serialization interfaces for Gen. Chapter 4 discusses implementations using tracing semantics for the Gen's Dynamic Modeling language and combinators. Chapter 5 discusses applications of serialization for data science, debugging, and performant computation.

## 1.2 Related Work

Several other probabilistic programs provide some serialization for their native data objects. However, Gen is a universal probabilistic program, and its design provides extra challenges that are not encountered with restricted modeling languages such as Stan. [3] Gen's model inference design provides users highly customizable data-driven proposals, and this thesis addresses these limitations when serializing *trace* objects. The serialization and deserialization implementations we show resembles the implementations of Gen's other interfaces using tracing semantics as shown in the Julia implementation, Gen.jl. Other probabilistic programming languages are universal but do not allow users to specifiy custom inference proposals. [7]

It is helpful to compare other variational methods to the amortized learning scheme used in Section 5.4. [18], [13], [11] train parameterized proposals against the target posterior using an EUBO objective, but do not consider training *inference algorithms* (whose internals may use neural proposals). Variational inference is another popular technique to approximate target distributions, however, it instead optimizes an ELBO objective that seeks to fit a model to a collection of observations whose behavior is "mode-seeking". [1] The EUBO objective fits the model "on average" according to the target distribution. Variational autoencoders [10] and the extended Importance Weighted Variational Autoencoders [2] use networks to parameterize distributions although this thesis does not use the reparameterization trick to take gradients.

3DP3 is a probabilistic generative model designed to express objects and their poses in scenes.[8] Model inference uses a renderer-in-the-loop to score likelihoods of proposed hypothesis in parallel. Since the latent space is high dimensional, techniques such as coarse-to-fine gridding proposals can prune hypothesis regions. Using neural proposals could help reduce the grid search over the latent space, speeding pose estimation.

# Chapter 2
# Background

This chapter provides an overview of Gen's design and inference algorithms needed used throughout the thesis. Section 2.1 describes the Gen probablistic programming language, its interfaces, and how implementations use tracing to support the specification. The second half of the chapter reviews several inference algorithms that will be relevant in the later parts of the thesis. Section 2.2 reviews importance resampling and several common improvements to reduce variance of sampled estimators. The final section provides an overview to recent work in approximate inference using EUBO gradient estimators. This framework applies can be applied to importance sampling and is thus amenable to variational optimization.

## 2.1 The Gen PPL

### 2.1.1 Generative Functions and Traces

To model normal code as generative processes with randomness, Gen's specification uses the abstraction known as *generative function* to define functions with an associated probability distribution. Generative functions are core to Gen's modeling language and inference libraries, and in implementation are usually exposed as native functions defined in a programming language as shown in Figure 2.1. Along with the output to the function, a generative function also produces an addressable dictionary of the sampled random choices which is useful both as a namespace and for hierarchical structure as we will soon describe. We restate relevant definitions as defined in [5].

**Definition 2.1.1** (Address Universe)**.** Let $A$ be a finite or countably infinite set of addresses. For each $a \in A$, let $V_a$ denote the domain of $a$, where $V_a$ is finite or countably infinite. A pair of $(A, V)$, which defines a set of addresses and their domains, is called an *address universe.*

**Example 2.1.1.** Let $A = \{a, b\}$, $V_a = \{0, 1\}$, and $V_b = \{1, 2, 3, 4, 5, 6\}$. Then the address universe consists of $a$ and $b$ along with their domains.

**Definition 2.1.2** (Choice dictionary)**.** A choice dictionary in address universe $(A, V)$ is a map $\tau : A_\tau \to \bigcup_{a \in A} V_a$ where $A_\tau$ is a finite subset of $A$ and where $\tau[a] \coloneqq \tau(a) \in V_a$ for all $a \in A_\tau$. Equivalently, $\tau$ is a finite set of pairs $\{(a_1, v_1) \ldots, (a_k, v_k)\}$ such that each $a_i \in A$ appears once and $v_i \in V_{a_i}$ for each $i$.

The choice dictionary is thus a map from addresses to selections of values from each address's domain. Selecting one value from each addresses domain is akin to sampling a random choice from event space of a random variable. There may be different possible choice maps consistent with an address universe and a choice map

```
@gen function line_model(x)
  y ~ normal(2*x+2,1)
end
```

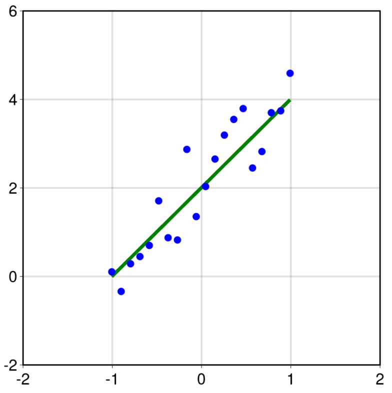


**Figure 2.1:** A Gen program modeling a line with noise.

need not use all addresses availabe from the address set. For an address universe $(A, V)$, let $B \subset A$ be a selection of addresses. Define $\mathcal{T}_B$ to be the collection of choice maps $\tau$ such that $A_\tau = B$. Similarly let $\mathcal{T}_B^*$ be the collection of choice maps such that $A_\tau \subset B$. Two choice dictionaries $\tau$ and $\sigma$ are said to be *disjoint* if $A_\tau \cup A_\sigma = \varnothing$ and let $\tau \oplus \sigma$ be the concatentation.

**Example 2.1.2** (Coin and Die Choicemap)**.** From Example 2.1.1, there are 21 possible choice maps in $\tau_A^*$.

$$\mathcal{T}_A^* = \begin{cases} \{\} \\ \{a \mapsto 0\}, \{a \mapsto 1\} \\ \{b \mapsto n\} \ 1 \leq n \leq 6 \\ \{a \mapsto 0, b \mapsto n\}, \{a \mapsto 1, b \mapsto n\} \ 1 \leq n \leq 6 \end{cases}$$

A generative function is a function and a probability distribution over choice maps, effectively recording the random choices produced inside of the function. These random choices can for example be generated by simple random variables - both discrete and continuous - or through composition by other generative functions. Drawn choices maps from the probability distribution and return values from the function create what is known as a trace. Traces act as samples from the generative function and are primarily used in inference.

**Definition 2.1.3** (Generative Function and Trace)**.** A generative function in address universe $(A, V)$ is a tuple $\mathcal{P} = (X, Y, p, f)$. $X$ is the argument type, $Y$ is the return type. For every $x \in X$, $p(\cdot; x) : \mathcal{T}_A^* \to [0, 1]$ is a *structured* and *well-behaved* probability density on $\mathcal{T}_A^*$. The function $f : \{(x, \tau) : \tau \in \text{supp}(p(\cdot; x))\} \to Y$ maps arguments and supported choice dictionaries to return values $y \in Y$. A *trace* from generative function $\mathcal{P}$ is a sample of the form $(\mathcal{P}, x, \tau)$ where $x$ are arguments and $\tau$ is the choice dictionary.

**Example 2.1.3.** The Gen program written in Figure 2.1 can viewed as a generative function. The generative function produces traces where $x \in \mathbb{R}$. The address universe is $\{y\}$ and $V_a = \mathbb{R}$, so $\tau$ is a choice mapping the only address available, y, to $\mathbb{R}$. The input and return types of $\mathcal{P}$ are $(X, Y) = (\mathbb{R}, \mathbb{R})$. The function $f : \mathbb{R} \times \tau_A \to [0, 1]$ maps $(x, \tau)$ to $\tau[y]$. Finally $p(\cdot; x)$ is defined as the Gaussian

$$p(\tau; x) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{(\tau[y] - (2x + 2))^2}{2}\right)$$

for $\tau \in \mathcal{T}_A$ and 0 otherwise.

We use *choice map* and choice dictionary interchangeably. As shown in [5], functions remain generative functions for certain compositions such as calling a two generative functions in sequence. This composition simplifies the description of a generative function into modular pieces and establishes hierarchical structures that resembles code-like execution. Gen requires that the composition of generative functions does not cause an address conflict (e.g. two functions addressing the same variable).

### 2.1.2 Modeling Languages and Generative Function Interfaces

To model generative processes in Gen, there are two default modeling languages suited for different use cases. The Dynamic Modeling Language (DML) provides users the ability to express models using nearly arbitrary code execution. For example, in the Julia version of Gen, Gen.jl, models can be written as functions with control flow constructs like loops, switch statements, and recursion. While highly dynamic, the inference algorithms can be slow. The Static Modeling Language (SML) is a restricted set of Gen with limited control flow, but amenable to static analysis to determine model structure ahead of time. Thus inference is specialized and is asymptotically faster than their analogs in the DML.

Gen decomposes inference computation into a minimal set of interfaces needed to support common operators such as evaluating log-likelihoods and updating importance weights as these are operations found in most algorithms. Both modeling languages support these interfaces in the form of generative functions and traces. Some functions produce new traces whereas others take in a trace and produce new traces with updated random chocies and log-likelihood scores. Algorithm 1 summarizes these methods.

### 2.1.3 Tracing as an Implementation

Tracing is a popular method used in a wide array of probabilistic programming languages and automatic differentiation systems. Tracing involves intercepting key points of execution to run *effect handlers* which are functions that perform operations opaque to the user. This method abstracts the model specification - which the user is responsible for - from the automated tooling needed to run generative models as native functions. Some of Gen's implementations use tracing to support the interfaces and intercept sample calls (e.g. with the $\sim$ syntax in Julia). These traced calls execute effect handlers under-the-hood to as shown in Figure 2.2 for Gen.jl.

**Algorithm 1** Gen Model Interfaces

```
• $t \leftarrow$ simulate(arguments: $x$, distribution : $\mathcal{P}$)
Generate a trace from the distribution.

• $t \leftarrow$ generate(arguments: $x$, distribution: $\mathcal{P}$, choices: $\mathcal{T}$ )
Returns a trace from a distribution that is consistent with passed choices.

• $t', \tau' \leftarrow$ update(arguments: $x$, trace: $t$, choices: $\tau$)
Update trace's arguments and choices consistent with the passed choices.
Returns a new trace and discarded choices from the old trace.

• $\tau \leftarrow$ choices(trace: $t$)
Return the choicemap.

• $\tau \leftarrow$ assess(arguments: $x$, distribution: $\mathcal{P}$, choices: $\tau$)
Return an importance score.
```

```
@gen function model()          function model():
  x ~ normal(0,1)                  x = trace_at(state, :x , normal, (0,1))
  y ~ submodel(x)                  y = trace_at(state, :x , submodel, (x,))
end                            end

@gen function submodel(x)      function submodel(x)
  z ~ normal(x,1)                z = trace_at(state, :z , normal, (x,1))
end                            end
```

**Figure 2.2:** Gen.jl uses program transformations to convert human-readable generative models to functions that enable tracing at the stochastic choices.

**Encapsulated and Untraced Randomness** Although the choice dictionary is defined to contain all random choices produced by the generative function, this does not mean a generative function must relegate all randomness to the choice dictionary. *Encapsulated randomness* is black-box randomness that is not accesible to the user. In practice, encapsulated randomness usually occurs as *untraced* randomness when users do not or are unable to use the tracing semantics in the language. For instance, any foreign calls to third-party libraries cannot immediately be traced and therefore any internal stochastic behavior is unaccounted for. Traces generated from a model containing untraced randomness can create variability in likelihood scores, choice maps, and even return values. Despite this Gen allows for *encapsulated* (untraced) randomness so long as the model satisfies several assumptions. [5]

## 2.2 Importance Sampling

Importance sampling is a widely-used technique for estimating expectations with respect to a target distribution by drawing samples from a proposal distribution.[17] Often this target distribution is not known nor can it be sampled from, so an auxilliary distribution known as a proposal is used to simulate samples.

**Sequential Importance Resampling** In practice traditional importance sampling can suffer due to high dimensionality or with ill-conditioned proposals. In high-dimensional spaces, the proposal distribution often struggles to adequately cover regions with significant probability mass in the target distribution. Consequently, a large number of low-weighted samples are obtained, resulting in poor estimation accuracy. Sequential importance *resampling* attempts to address this issue by resampling from the set of weighted samples produced by importance sampling and thus prevents particles from degenerating. Algorithm 2 shows the algorithm.

**Conditional Importance Sampling** Conditional importance sampling is a boostrap algorithm that uses a conditional distribution as a proposal. Whereas SIR resampled particles from a collection of proposed particles, CIS starts with one particle and produces a collection of new particles as showin in Algorithm 3.

**Algorithm 2** Sampling Importance Resampling (SIR)

```
procedure SIR($N$, $p(\mathbf{x})$, $q(\mathbf{x})$)
    for $i \in [N]$ do                                ▷ Sample $N$ particles from proposal
        $\mathbf{x}_i \sim q(\mathbf{x})$
        $w_i \leftarrow p(\mathbf{x}_i)/q(\mathbf{x}_i)$
    end for
    $W \leftarrow \sum_{i=1}^{N} w_i$                                   ▷ Normalize Weights
    $w_i \leftarrow w_i/W$ for $i \in [N]$
    // Resampling: Draw samples with replacement according to the weights
    for $i = 1$ to $N$ do
        $\mathbf{x}_i \sim$ CATEGORICAL($\{w_i\}_{i=1}^{N}$)                ▷ Resample particles
    end for
    return $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\}$, $\{w_1, w_2, \ldots, w_N\}$      ▷ Return collection of particles
end procedure
```

## 2.3 Amortized Inference

Bayesian models provide expressive model structure, but often at the cost of slow exact inference. Inference algorithms such as importance sampling or Markov Chain Monte Carlo can approximate target distributions, but the choice proposal heavily influences convergence speed. Amortized inference is a broad set of methods to make Bayesian inference tractable by using a parameterized distribution - often in the form

**Algorithm 3** Conditional Importance Sampling (CIS)

```
procedure CIS($N$, $x$, $p(\mathbf{x})$, $q(\mathbf{x})$)
    $j \sim$ CategoricalUniform(N)                ▷ Sample uniformly from $[n]$
    $w_j \leftarrow p(\mathbf{x}_j)/q(\mathbf{x}_j)$
    for $i \in [N]$ and $i \neq j$ do
        $\mathbf{x}_i \sim q(\mathbf{x})$
        $w_i \leftarrow p(\mathbf{x}_i)/q(\mathbf{x}_i)$
    end for
    return Sample set $\{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N\}$ with weights $\{w_1, w_2, \dots, w_N\}$
end procedure
```

of a neural network - to provide fast inference approximations and instead *amortize* the cost of inference to training time. Recent work in [11] has explored how to train LSTM architectures on data averaged posteriors for fast, approximate inference. In [11], training involves sampling from the model joint distribution and conditioning recurrent neural proposals on observations and ancestor samples to generate distributional parameters in a proposal with the same shape and support as the model. This approach imposes strong (and, often, wrong) assumptions on the latent dependencies induced by inference, exacerbating training times and potentially leading to poor inference approximation performance.

We highlight another amortized learning approach which uses a similar evidence upper bound (EUBO) objective as in [11], but utilizes inference approximations based on the variational importance sampling family [2]. We will illustrate how to combine serialization with this training method to amortize the cost of inference targets with fast approximated neural likelihoods for slow likelihood models. The final learned approximations learn the target and can be re-used for different observations $x$ to generate good approximations to $p(z|x)$.

### 2.3.1 EUBO Objective and Gradient Estimators

Let $p(x, z)$ be the joint distribution for latents $z$ and observations $x$. The goal is to determine an approximation for the posterior $p(z|x)$ under the EUBO objective:

$$D(p(z|x)||Q_\theta(z;x)) = \int p(z|x) \log \left[ \frac{p(z|x)}{Q_\theta(z;x)} \right] \mathrm{d}z$$

where $Q_\theta(z;x)$ is a parameterized distribution, and the learning $Q_\theta(z;x)$ arises from minimizing the divergence. In contrast to the ELBO which produces approximations that are said to be "mode-seeeking", the EUBO objective is a *model* average and tends to promote "mass-covering" behavior.[9] This is beneficial in the case of approximating $p(z|x)$ as the learned model should work well on average for multiple observations $x$.

As opposed to inference compilation and similar learning schemes, we consider $Q_\theta(z;x)$ to be the distribution of an inference algorithm *itself*. In particular let $Q_\theta(z;x)$ be parameterized SIR with *scorer* $Q_s(z;x)$ and *proposal* $Q_p(z;x)$ (we suppress dependence on $\theta$). While SIR does output one $z$ given observations $x$, internally the sampler proposes a collection of particles and samples one candidate proportional to the importance weights. The accessible distribution is of the form $Q_\theta(z^*, I, z_1, \ldots z_n; x)$ where $\{z_i\}_{i=1}^n$ is the particle collection and $I$ is the index satisfying $z^* = z_I$, but note the EUBO objective requires the marginal $Q_\theta(z; x)$ instead. We estimate the the value of $Q_\theta(z; x)$ (and its gradient) using recent work leveraging the pseudo-marginal estimators introduced in [12].

A *meta* distribution describes the *auxilliary randomness* of $Q$ to be marginalized out. For SIR, this is the distribution $Q^{meta}(I, z_1, \ldots, z_{I-1}, z_{I+1}, \ldots z_n; x)$. *Meta-inference* then uses $Q^{meta}$ to propose values for the auxillliary randomness. Here we use conditional importance sampling as defined in Algorithm 3. Define the unbiased estimator $\hat{Q}(z; x)$ for the marginal $Q(z; x)$ as

$$\hat{Q}(z; x) = \frac{1}{K} \sum_{k=1}^{K} \frac{Q(z, I^{(k)}, z_1^{(k)}, \ldots z_{I^{(k)}-1}^{(k)}, z, z_{I^{(k)}+1}^{(k)} \ldots, z_n^{(k)}; x)}{Q^{meta}(I^{(k)}, z_1^{(k)}, \ldots, z_{I^{(k)}-1}^{(k)}, z_{I^{(k)}+1}^{(k)}, \ldots z_n; x)} \tag{2.1}$$

The following argument shows that Equation 2.1 is unbiased.

**Proposition 2.3.1.** Let $p(x, y)$ be a distribution. For samples $y_k$ drawn from $q(y)$,

$$\hat{p}(x) = \sum_{k=1}^{K} \frac{1}{K} \frac{p(x, y_k)}{q(y_k)}$$

is an unbiased estimator of the marginal $p(x)$.

*Proof.*

$$\mathbb{E}\left[\frac{1}{K} \sum_{k=1}^{K} \frac{p(x, y)}{q(y)}\right] = \mathbb{E}\left[\frac{p(x, y)}{q(y)}\right] = \int q(y) \frac{p(x, y)}{q(y)} \, \mathrm{d}y = p(x)$$

□

Similarly, we can use a REINFORCE-like estimate for the gradient $\nabla_\theta Q_\theta(z; x)$.[19] Assuming $\nabla_\theta$ and the expectation commute, the gradient with respect to the parameters is

$$\begin{aligned} \nabla_\theta E\left[\log \frac{p(x, z)}{Q_\theta(z; x)}\right] &= \nabla_\theta \int p(x, z) \log \left[\frac{p(z|x)}{Q_\theta(z; x)}\right] \\ &= -\int p(x, z) \nabla_\theta \log Q_\theta(z; x) \end{aligned}$$

# Chapter 3
# Serialization Interfaces

In this chapter we introduce two new interfaces for serialization of traces. We first motivate the discussion in Section 3.1 by exploring why standard serialization tools and the existing Gen interfaces are not enough to read traces into memory. In the second half, we provide a specification for the interfaces.

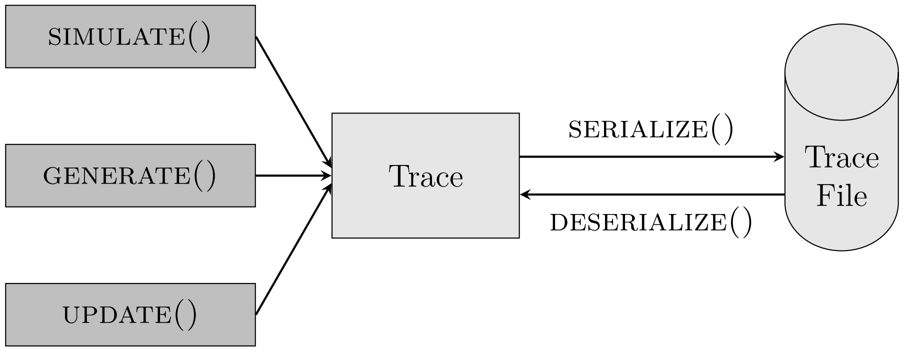


**Figure 3.1: Overview of Serializing Traces** Users call inference methods such as SIMULATE to produce traces. Calling the serialization methods stores a trace to disk which can later be read back into memory for further inference tasks.

## 3.1 Limitations of Existing Methods

A trace is defined as $(\mathcal{P}, x, \tau)$ where $\mathcal{P}$ is the generative function. A Gen implementation can represent generative functions as native functions in the language of choice, and trace implemenations may hold data objects ephemeral to the runtime. For example, both the Dynamic and Static Modeling Languages hold function pointers in each trace (and subtraces for the DML). Thus serializing a trace would entail serializing the function identities, which many standard serialization tools cannot reliably save. [1] We direct our attention to problems with serializing functions, but trace implementations may have other runtime-specific attributes and similar roadblocks will apply.

Nearly all languages associate functions with arbitrary memory addresses only defined at runtime and are thus temporary - deserialization during a fresh runtime will fail because the pointer will likely reference invalid memory. Moreover, even if a function identity could be deserialized, there are still several pitfalls that create unintended side-effects.

- **Changes in Function Definition** If the function used to generate the trace changes slightly (e.g. adding print statements or changes in logic), then upon

[1]Some tools like Python's *dill* can serialize functions and simplify this problem.

deserialization two new functions could be added. This is typical in packages that support function serialization. The runtime does not recognize that the new function is simply a redefinition of the old one and may in fact override it.

- **Allocating Foreign Functions** Similar to the first issue, the trace may contain a function that the user was not aware about. This clutters the namespace and may create security vulnerabilities that a malicious party can exploit. Consider the following example. Suppose trace $t_{trust}$ is modified by replacing its generative function with malicious code. Then on a call to UPDATE, the malicious code executes on a user's machine.

- **Large File Sizes** Serializing functions can create consume a large memory footprint as it requires saving the definition. Moreover, if function is defined as a closure, it must save captured variables defined in the lexical scope of the closure.

**First Attempt** This previous discussion suggests that the function pointer should be discarded and the rest of the trace should be serialized (e.g. arguments and choice map). A possible scheme consists of two steps. First serialize all data objects of the trace except the function. Upon reading the contents back to disk, the trace is constructed by assigning manually the correct generative function which produced the trace. This unfortunately does not work since any *subtraces* serialized would have also required discarding their function pointers. The user would need to know exactly which functions were in the callstack to reattach *all* function pointers. This becomes very untenable for all but the simplest generative functions. A complicated, highly dynamic model would produce various traces with different address structures and would requiring saving away the entire call stack of traced calls.
**Second Attempt - Using GENERATE** Another alternative is to utilize the existing interfaces to deserialize. In particular, the GENERATE method accepts a choice map and produces a fresh trace (with associated function pointers). The choicemap is used to constrain all the random choices made by callees, so any subcall with a matching address is automatically constrained by the corresponding address the provided choicemap. This method also fails because GENERATE does not guarantee the effects of untraced randomness are ignored and can erroneously produce incorrect scores. There is still another problem even if the score can be correctly serialized - it is self-defeating to call GENERATE and incur the cost of executing the generative function again. This will motivate an implementation of DESERIALIZE that defers the cost of running the generative function until it is absolutely necessary.

**Example 3.1.1** (Untraced randomness in GENERATE)**.** Figure 3.2 shows a program with untraced randomness. The table shows all possible traces if a call to SIMULATE sampled $x = 0.5$, and each has a score of $2\ln(1/2) \approx -1.386$. If $\sigma = \{y \mapsto 0, z \mapsto 0\}$ was saved out, then calling MODEL.GENERATE($\sigma$) does produce a trace $\mathbf{t}$ where $\mathbf{t}.\tau = \sigma$. However, during the execution of GENERATE is is possible for $x \neq 0.5$ and as a result change the score. For example, if $x = 0.25$, then the "deserialized" trace has score $2\ln(3/4) \approx -0.575$ which is different from the old one.

```
@gen function model()
  x = rand() # Untraced
  y ~ bernoulli(x)
  if y == 0
    z ~ bernoulli(x)
  else
    z ~ bernoulli(1-x)
  end
  return z
end
```

| $\tau$ | $p(\tau)$ | Score |
|---|---|---|
| $\{y \mapsto 0, z \mapsto 0\}$ | 0.25 | -1.386 |
| $\{y \mapsto 0, z \mapsto 1\}$ | 0.25 | -1.386 |
| $\{y \mapsto 1, z \mapsto 0\}$ | 0.25 | -1.386 |
| $\{y \mapsto 1, z \mapsto 1\}$ | 0.25 | -1.386 |

**Figure 3.2**

## 3.2 Serialization Operations for Traces

Gen exposes generative functions and traces as *abstract data types* (ADTs) with supported operations listed in Listing 1. [5] ADTs abstract away the internals and implementation of an object to create modular functionality which increases the ease of compositionality and design of code. The same abstractions can be extended to support two primitive operations called SERIALIZE and DESERIALIZE for trace ADTs.

We first restrict to generative models that do not use a proposal or contain encapsulated randomness to simplify the discussion, but we later drop these assumption. Let $\mathcal{P} = (X, Y, p, f)$ be a generative function where $X$ is the argument type and $f$ maps arguments $x \in X$ to outputs in $Y$. Furthermore, $\mathcal{P}$ produces traces $\mathbf{t} = (\mathcal{P}, x, \tau)$ such that $\mathbf{t} \in \text{supp}(p(\cdot; x))$. Recall that $p$ satisfies the following condition [5]:

**Definition 3.2.1** (Structured Probability Distribution on Choice Dictionaries)**.** A probability distribution $p$ on choice dictionaries is called *structured* if for all $\tau$, $\tau' \in$ supp(p) either $\tau = \tau'$ or $\tau[a] \neq \tau'[a]$ for some $a \in A_\tau \bigcap A_{\tau'}$.

Serializing a trace $\tau$ to disk involves finding some *encoding* that is amenable to storage. Define $\langle x \rangle$ as the encoding of an object $x$ which represents a string description of the object. The map $\langle \cdot \rangle$ abstracts away details of serialization for an object and is assumed to exist for primitive types (e.g. numbers and strings).[2]. The encoding can later be read to reconstruct the object. We use the syntax $\langle x, y \rangle$ as a shorthand for the concatenation of encodings.
**Assumption**: Elements of an address set $A$, address values $\bigcup V_a$ , argument type $X$, output type $Y$, and real numbers have encodings. In light of the challenges with saving out functions stated in Section 3.1, $\langle \mathcal{P} \rangle$ is not necessarily available (but $\mathcal{P}$ is).

**Definition 3.2.2** (Encoding of a Choice Dictionary)**.** Let $\tau : A_\tau \to \bigcup_{a \in A} V_a$ be a choice dictionary. Then $\langle \tau \rangle = \langle a_1, \tau[a_1], \ldots, a_k, \tau[a_k] \rangle$ is its encoding. For disjoint choice dictionaries $\tau$ and $\sigma$, $\tau \oplus \sigma$ has encoding $\langle \tau \oplus \sigma \rangle = \langle \tau, \sigma \rangle$

[2]Exactly which objects have an encoding is beyond scope, but in practice there are many tools that handle primitives (e.g. floats and immutable structs) to produce binary descriptions. Thus we delegate how to create and decode encodings to other tools.

Definition 3.2.2 is one choice of encoding choice dictionaries, but it is a convenient one for modularity. This is useful when describing choice dictionaries concatenated from mupltiple generative functions. Now we use an example model written in Gen.jl's Dynamic Modeling Language to illustrate the operations.

```
@gen function model()
  cow ~ uniform(0,1)
  if x < 0.5
    moo ~ bernoulli(0.1)
  else
    jump ~ normal(0,1)
  end
  return 2*cow
end
```

**Serialize Operation** The operation $\mathbf{t}.\textsc{serialize}()$ takes in a trace $\mathbf{t} = (\mathcal{P}, x, \tau)$ and first calculates the return value $y = f(x, \tau)$ using the return function of $\mathcal{P}$. The output is the encoding $\langle x, y, \tau \rangle$.

**Example** Suppose $\mathbf{t} = (\textsc{model}, \perp, \{\text{cow} \mapsto 0.48, \text{moo} \mapsto 1\})$. The return value is $y = 0.96$ and the encoding is $\langle \perp, 0.96, \{\text{cow} \mapsto 0.48, \text{moo} \mapsto 1\} \rangle$.

**Deserialize Operation** Given an encoding of a trace $e = \langle x, y, \tau \rangle$, $\mathcal{P}.\textsc{deserialize}(e)$ decodes $\langle x \rangle$, $\langle y \rangle$, and $\langle \tau \rangle$ to produce the trace $\mathbf{t} = (\mathcal{P}, x, \tau)$. Here it is assumed that $\mathcal{P}$ is a generative function that produced the trace or in other words $p(\tau; x) > 0$ and $y = f(\tau, x)$.

**Example** Calling $\mathcal{P} = \textsc{model}$ on the previous encoding reconstructs the trace $\mathbf{t} = (\textsc{model}, \perp, \{\text{cow} \mapsto 0.48, \text{moo} \mapsto 1\})$. The encoding $\langle \perp, -1.0, \{\text{cow} \mapsto -0.5, \text{moo} \mapsto 1\} \rangle$ does not lie in the support of $p$ and so deserialization has undefined behavior.

## 3.3 Compositionality of Interfaces

Both SERIALIZE and DESERIALIZE are compositional, and we highlight this compositionality for two ubiquitous control flows: *sequencing* and *branching.* [5]

**Sequencing Generative Functions** Two generative functions $\mathcal{P}_1 = (X_1, Y_1, p_1, f_1)$ and $\mathcal{P}_2 = (X_2, Y_2, p_2, f_2)$ can be called in sequence to define a third generative function $\mathcal{P}_3$. The argument $x$ to $P_3$ is the same as $\mathcal{P}_1$'s. First a choice map is drawn from $p_1(\cdot; x)$ and the return value $f_1(\tau|_{A_1}, x)$ is then passed as arguments to $\mathcal{P}_2$. The remaining portion of the choice map is drawn according to the distribution $p_2(\tau|_{A_2}; f_1(x, \tau|_{A_1}))$.

The construction is summarized below.

$$\begin{aligned}
&X_3 \coloneqq X_1\\
&Y_3 \coloneqq Y_2\\
&f_3(x,\tau) \coloneqq f_2(f_1(x,\tau|_{A_1}),\tau|_{A_2})\\
&p_3(\tau;x) \coloneqq p_1(\tau|_{A_1};x)p_2(\tau|_{A_2};f_1(x|\tau_{A_1}))
\end{aligned}$$

Note this construction is only valid if $A_1$ and $A_2$ are disjoint as per Gen's assumptions. Now suppose $\mathcal{P}_3$ is evaluated at $x$ and returns the trace $\mathbf{t}_3 = (\mathcal{P}_3, x, \tau_3)$ with return value $y_3 \in Y_3$. By definition, the corresponding traces from $\mathcal{P}_1$ and $\mathcal{P}_2$ are $\mathbf{t}_1 = (\mathcal{P}_1, x, \tau_3|_{A_1})$ and $\mathbf{t}_2 = (\mathcal{P}_2, f(x,\tau_3|_{A_1}), \tau_3|_{A_2})$ respectively. Moreover, the encodings produced by calling SERIALIZE on $\mathbf{t}_1$ and $\mathbf{t}_2$ are

$$\begin{aligned}
&\langle x, f(x,\tau_3|_{A_1}), \tau_3|_{A_1}\rangle\\
&\langle f(x,\tau_3|_{A_1}), f(f(x,\tau_3|_{A_1}),\tau_3|_{A_2}), \tau_3|_{A_2}\rangle
\end{aligned}$$

Thus it is clear what $\langle \mathbf{t}_3\rangle$ should be:

$$\langle x, f(f(x,\tau_3|_{A_1}),\tau_3|_{A_2}), \tau_3|_{A_1}, \tau_3|_{A_2}\rangle = \langle x, f(f(x,\tau_3|_{A_1}),\tau_3|_{A_2}), \tau_3\rangle$$

which is equivalent to the encoding produced by $t_3$.SERIALIZE(). The DESERIALIZE operation works similarly compositional and uses the structure property in Definition 3.2.1. Given an encoding $\langle x, y, \tau\rangle$ of a trace $\mathbf{t}_3$ from $\mathcal{P}_3$, there exists by definition some $A_1$ such that $\tau|_{A_1} \in \text{supp}(p_1(\cdot;))$. We recalll the following theorem due to Cusamano-Towner [5].

**Proposition 3.3.1.** For a structured probability density $p$ on choice dictionaries, and some $\sigma \in \mathcal{T}_A^*$, if $\tau_1 = \sigma|_{B_1} \in \text{supp}(p)$ and $\tau_2 = \sigma|_{B_2} \in \text{supp}(p)$ for some $B_1$, $B_2$ then $\tau_1 = \tau_2$.

Since $p_1$ is a structured probability distribution, by Proposition 3.3.1 $A_1$ is unique. Define $A_2 = A_\tau \backslash A_1$. Then the encodings produced by $\mathcal{P}_1$ and $\mathcal{P}_2$ were $e_1 = \langle x, f(x,\tau_{A_1}), \tau_{A_1}\rangle$ and $e_2 = \langle f(x,\tau_{A_1}), f(f(x,\tau_{A_1}),\tau_{A_2}), \tau_{A_2}\rangle$. Finally $\mathcal{P}_1$.DESERIALIZE $= (\mathcal{P}_1, x, \tau_{A_1})$ and $\mathcal{P}_2$.DESERIALIZE $= (\mathcal{P}_2, f(x,\tau_{A_1}), \tau_{A_2})$. Since $\mathcal{P}_1$ executes first before $\mathcal{P}_2$, the combined trace is $(\mathcal{P}_3, x, \tau_{A_1} \oplus \tau_{A_2})$ or equivalently $\mathbf{t}_3$. Note that for both SERIALIZE and DESERIALIZE, some arguments and return values ($f(x,\tau_{A_1})$ and $f(f(x,\tau_{A_1}),\tau_{A_2})$) of the two constituient traces were dropped to encode and decode $\mathbf{t}_3$. We later show that it is useful in implementations to store these intermediary values and necessary when dealing with encapsulated randomness.

**Conditional Branching Generative Function** A conditional branch between two generative functions takes in a boolean value as well as the two arguments of the respective generative functions. Only one generative function is called depending on the boolean value. Let $(b, x_1, x_2)$ be the boolean and arguments. Then $\mathcal{P}_3$ is defined as

$$\begin{aligned}
X_3 &:= \{0,1\} \times X_1, \times X_2 \\
Y_3 &:= Y_1 \cup Y_2 \\
p_3(\tau; (b, x_1, x_2)) &:= \begin{cases} p_1(\tau; x_1) & \text{if } b = 0 \\ p_2(\tau; x_2) & \text{if } b = 1 \end{cases} \\
f_3((b, x_1, x_2), \tau) &:= \begin{cases} f_1(x_1, \tau) & \text{if } b = 0 \\ f_2(x_2, \tau) & \text{if } b = 1 \end{cases}
\end{aligned}$$

Again, SERIALIZE remains compositional. If $\mathbf{t} = (\mathcal{P}_3, x, \tau)$, the call $\mathbf{t}$.SERIALIZE() produces the encoding

$$\langle \mathbf{t} \rangle = \begin{cases} \langle (0, x_1, x_2), f_1(x_1, \tau), \tau \rangle & \text{if } b = 0 \\ \langle (1, x_1, x_2), f_2(x_2, \tau), \tau \rangle & \text{if } b = 1 \end{cases}$$

which exactly equivalent to encoding the constituient traces and selecting one based of the value $b$. To deserialize $\langle (b, x_1, x_2), y, \tau \rangle$, the value of $b$ determines whether to call $\mathcal{P}_1$.DESERIALIZE($\langle x_1, y, \tau \rangle$) or $\mathcal{P}_2$.DESERIALIZE($\langle x_2, y, \tau \rangle$).

## 3.4 Internal Proposals and Encapsulated Randomness

In the previous section, we assumed that the only source of randomness eminated from the choice dictionary and this greatly simplified deserialization. In fact without encapsulated randomness, it is actually sufficient to reconstruct traces with the GENERATE approach mentioned in Section 3.1. Intuitively, the choice dictionary of the serialized trace serves as a record of the call graph that GENERATE can "follow" to reproduce the trace. However, this fails with untraced randomness, and so the interfaces need to be modified to account for these types of generative functions.

Gen, likewise, permits the use proposal distributions for inference. What a generative function uses a proposal distribution for can vary. For example, inference algorithms can use a proposal distribution to transition between states as done in Metropolis-Hastings or SMC. [4] A proposal distribution can furthermore use data to better explore regimes of latent space when approximating posterior distributions. Thus the proposal distribution is a central part of Gen's design and every generative function comes with a default one known as an internal proposal. We restate a couple of definitions. [5]

**Definition 3.4.1** (Encapsulated Randomness)**.** A generative function $\mathcal{P}$ with encapsulated randomness is a tuple $\mathcal{P} = (X, Y, p, f, q, \Omega, \mathring{p}, \mathring{q})$ where $(X, Y, p, f, q)$ is a generative function with an internal proposal family, $\mathring{p}$ and $\mathring{q}$ are families of probability densities on $\omega \in \Omega$ such that $\mathring{p}(\omega; x, \tau) > 0$ if and only if $\mathring{q}(\omega; x, \tau) > 0$ for all $x, \tau$

such that $p(\tau; x) > 0$. A trace $\mathbf{t}$ of such a generative function is a tuple $(\mathcal{P}, x, \tau, \omega)$ where $\omega \in \Omega$ satisfies $\mathring{p}(\omega; x, \tau) > 0$. The return function $f$ is a function of $x$, $\tau$, and $\omega \in \Omega$.

The joint distribution $p(\tau; x)\mathring{p}(\omega; x, \tau)$ governs the choice dictionaries and encapsulated randomness from the generative function. Even with a fully specified trace, invocations of the generative function may not be deterministic as it also depends on the encapsulated randomness produced. This becomes problematic when performing inference using the log-scores produced by GENERATE and REGENERATE. These log-scores are a function of estimators $\xi(x, \tau, \omega)$ which approximate the value $p(\tau; x)$. Intuititely, modifying the serialization interfaces requires encoding the encapsulated randomness.

**Storing $\omega$ to disk** In some cases, it is possible to store $\omega$ along with the serialized trace. Functions written in the modeling language can make untraced calls to known random sources (e.g. RAND()) and store the values. However, this is limited to untraced randomness that can be directly observed, which is self-defeating. It would be more beneficial to trace random calls rather than tracking them separately as untraced calls. More typically, a user may need to call black-box stochastic functions in their model and therefore has no knowledge of $\omega$. Although $\omega$ may not be known, it is still possible to guarantee serialization of encapsulated randomness by observing that the return values and log-likelihood scores are functions of $\omega$. This fact is exploited to modify the interfaces. Again, we guide the remainder of the section using an example of a model with untraced randomness.

```
@gen function model(sigma)
  x = rand() - 0.5
  v ~ normal(x^2, sigma)
  return v
end
```

**Serialization with Encapsulated Randomness** Given trace $t = (\mathcal{P}, x, \tau, \omega)$ first compute the return value $y = f(x, \tau, \omega)$ and density estimate $\xi(x, \tau, \omega)$. The output to $\mathbf{t}$.SERIALIZE() is the encoding is $\langle x, y, \tau, \xi(x, \tau, \omega)\rangle$.

**Example** Consider the unbiased estimator

$$\xi(x, \tau, \omega) = \frac{p(\tau, x)\mathring{p}(\omega; x, \tau)}{\mathring{q}(\omega; x, \tau)} \tag{3.1}$$

using some proposal $q$ and its corresponding encapsulated randomness $\mathring{q}(\omega; x, \tau)$. The call RAND() is an untraced call that produces a value in $[0, 1)$. Suppose the argument `sigma` $= 1.0$ and $x = 0.1$, then $\mathbf{t} = (\mathcal{P}, 1.0, \{\mathrm{v} \mapsto 0.02\})$ is a possible trace.[3]. The encoding is $\langle 1.0, 0.02, \{\mathrm{v} \mapsto 0.02\}, \xi(1.0, \{\mathrm{v} \mapsto 0.02\}, 0.1)\rangle$.

[3] $(0.1)^2$ plus noise

**Deserialization with Encapsulated Randomness** For this operation, the encoding $e = \langle x, y, \tau, \xi \rangle$ must come from a valid trace. In other words, $p(\tau; x) > 0$ and there exists an $\omega \in \Omega$ such that $y = f(x, \tau, \omega)$. Since such an $\omega$ is assumed to exist, $\mathcal{P}$.DESERIALIZE$(e)$ ouptuts $(\mathcal{P}, x, \tau, \omega)$.

Note that there may be more than one possible value of $\omega$ that satisfies the return value constraint *and* evaluates to the same value $\xi$ as in Equation 3.1. It is possible to treat the equivalence class over all $\omega$ that satisfies these two relations as the returned encapsulated randomness, but as we will show it is sufficient to consider only *one* representative element of the class. Therefore, this interface only guarantees that some $\omega$ satisfies the relations and does not guarantee any particular value.

**Example** Given the encoding from previous example, there are two consistent traces: $(\mathcal{P}, 1.0\{v \mapsto 0.02\}, 0.1)$ or $(\mathcal{P}, 1.0, \{v \mapsto 0.02\}, -0.1)$. DESERIALIZE will output one of them.

We have shown the minimum requirements needed to store a trace onto disk and correctly deserialize. Although any implementation must encode the four values into a file, each implementation is at the liberty to add auxilliary information to the encoding to help with deserialization. This is especially helpful for composed generative functions where callees may also be serialized. In Chapter 4, we show an example for the Dynamic Modeling Language.

### 3.4.1 Compatibility with Gen's Interfaces

The only modification to the encodings was to append the score estimates. Since $\omega$ is thrown out in the encoding, it may seem that deserialization requires finding a consistent $\omega' \in \Omega$. This is just as intractible as knowing $\omega$ itself for black-box functions. Without $\omega$, would it be possible to pass a *deserialized* trace into the other interfaces and guarantee that the importance scores remain valid? Note that there are only three existing interfaces that use a trace as input: LOGPDF, UPDATE, and REGENERATE. It is clear that if we save out the score estimate $\xi$, calling LOGPDF on the deserialized trace simply returns $\log \xi$. For the other two, $\omega$ only influences the value of the log-weight score [4]. This relinquishes the need to know $\omega$ in the first place, and *only* save $\xi$. Similar to Section 3.3, analogous arguments hold for combining generative functions together with encapsulated randomness.

## 3.5 Combinators

Gen exposes a set of combinators to compose generative functions in common ways. [5] Table 3.1 lists the existing combinators. Note that these combinators are very similar to the *sequencing* and *branching* compositions in the previous chapter and

[4]For example, UPDATE computes $\log(\xi(x', \tau', \omega')/\xi(x, \tau, \omega))$. The new choice dictionary is only a function of the original choice map and the proposed choice dictionary. Refer to [5]

moreover are compatible with both interfaces. Each combinator has a straightforward way to serialize, and all encodings take essentially the same form.

| **Combinator** | **Function** |
|---|---|
| Map | Broadcast $\mathcal{F}$ over a vector of inputs |
| Unfold | Apply $\mathcal{F}$ in sequence. Pass the return value of one application as input to the next. |
| Recurse | Call $\mathcal{F}$ to produce inputs, recurse on each input using $\mathcal{G}$, and combine outputs with $\mathcal{H}$. |
| Switch | Multiplexes $\{F_n\}_{n=1}^N$ and over vector of input and select one $n$. |

**Table 3.1:** Combinators composing generative functions to produce a new generative function.

**Map** Given a generative function $\mathcal{P}$, the combinator produces a new generative function $\mathcal{P}'$ that takes a vector of inputs $(x_1, \ldots, x_n) \in \bigcup_{m=0}^{\infty} X^m$ and produces a vector of outputs. Each call to $\mathcal{P}$ with arguments $x_i$ creates a choice map $t_i$ within a namespace and denote this by $\textsc{namespace}(\tau_i, i)$ for $i \in [n]$. A trace of $\mathcal{P}'$ takes the form $\tau' = \bigoplus_{i=1}^{n} \textsc{namespace}(\tau_i, i)$. Store

$$\langle x_1, \ldots x_n, y_1, \ldots, y_n, \tau', \xi_1 + \ldots + \xi_n \rangle$$

On deserialization, construct the trace

$$(\mathcal{P}', (x_1, \ldots x_n), \tau', \omega)$$

where $\omega$ is consistent.

**Unfold** Let $\mathcal{P}$ be a generative function whose argument type equals the return type, or $X = Y$. Then $\mathcal{P}'$, the unfold combinator of $\mathcal{P}$, takes a pair $(x_0, n)$ and produces a sequence of outputs

$$\begin{aligned} y_1 &\coloneqq f(x_0, \tau_1, \omega_1) \\ &\vdots \\ y_n &\coloneqq f(y_{n-1}, \tau_{n-1}, \omega_{n-1}) \end{aligned}$$

Now consider a trace $\mathbf{t} = (\mathcal{P}', (x_0, n), \tau, \omega)$ where $\tau = \bigoplus_{i=1}^{n} \textsc{namespace}(\tau_i, i)$. The encoding is

$$\langle x_0, n, y_1, \ldots, y_n, \tau, \xi_1 + \ldots + \xi_n \rangle$$

**Switch** The Switch combinator is an extension to branching. The encoding is similar to that in Section 3.3.

**Recurse** Let $\mathcal{P}_p$ and $\mathcal{P}_r$ be the production and reduction generative functions. The Recurse combinator takes $\mathcal{P}_p$ and $\mathcal{P}_r$ and a produces a third function $\mathcal{P}'$ defined by

$$\begin{aligned} v, x_1, \ldots, x_n &\coloneqq f_p(x, \tau_p, \omega_p) \\ y_i &\coloneqq f'(x_i, \tau_i', \omega_i) \text{ for } i \in [n] \\ y &\coloneqq f_r(v, y_1, \ldots, y_n, \tau_r, \omega_r) \end{aligned}$$

with base case

$$\begin{aligned} v &\coloneqq f_p(x, \tau_p, \omega_p) \\ y &\coloneqq f_r(v, \tau_r, \omega_r) \end{aligned}$$

The encodings are $\langle x, y, \tau_r \oplus \bigoplus_{i=1}^{n} \tau_i' \oplus \tau_r, \xi_p + \xi_1' + \ldots \xi_n' + \xi_r \rangle$ if $n \geq 1$ and $\langle x, y, \tau_p \oplus \tau_r, \xi_p + \xi_r \rangle$ for $n = 0$.

# Chapter 4
# Serialization Implementations for Modeling Languages

The previous chapter introduced two new interfaces for serializing traces and laid out the minimal requirements for correct serialization, namely storing arguments, return values, the choice map, and score. This chapter focuses on implementing the interfaces for Gen's different modeling languages and combinators. There are two default modeling languages - the Dynamic Modeling and the Static Modeling Language, and each exposes a trace object whose internals are abstracted away from th user. We provide implementations for the Dynamic Modeling Language and the combinators using Gen's tracing semantics. Tracing is a common technique in probabilistic programming languages to intercept calls to randomness. *Traced* calls invoke effect handlers under-the-hood which store information such as the random choices and probability scores, and this process remains opaque to the user. In Section 4.1, we give an overview for the Dynamic Modeling Language and the combinators. The second half of the chapter focuses on making "encodings" more concrete and considers different formats on disk.

**Restrictions** In general, it is possible to write functions with untraced randomness that produce traces which cannot be deserialized correctly for later inference. However, Gen already places restrictions on the modeling language that coincide with these paint points. There are two restrictions that are relevant for deserialization. [5] *Restricted use of untraced randomness*: Untraced randomness cannot directly influence control flow. This forbids, for example, an untraced value to decide which branch of an if-else to take. Furthermore, this also restricts influence on the address support. *Restricted use of mutability*: Arguments to generative functions may not change. Figure 4.1 shows an example. We assume from this point on that all models written abide by the modeling language restrictions.

```
@gen function()
  x ~ uniform(0,1)
  if x < 0.5
    y ~ normal(0,1)
  end
  if rand() < 0.5
    z ~ normal(0,2)
  end
end
```

**Figure 4.1: An invalid Gen DML model** The first branch is legal, but the second branch uses the native RAND call which is untraced and therefore not permitted in the language.

## 4.1 Dynamic Modeling Language and Combinators

We explain now how to implement SERIALIZE and DESERIALIZE using the tracing semantics of the Dynamic Modeling Language. While implementing SERIALIZE will be straightfoward, creating an efficient DESERIALIZE implementation requires some care. First we explain a slow version that reuses the same tracing logic as the other interfaces, and later show how a minor change to can amortize deserialization cost over inference instead.

A DML trace is implemented as a recursive data structure that uses a dictionary-like container to map addresses to *subtraces*. The trace also stores the arguments, return value, generative function identity, and score. By convention, any subtrace has its own namespace [1]. It is useful to think of the trace as a call graph but only for generative functions calls - it is a history of all the callees invoked.

The pseudocode for SERIALIZE is shown in Algorithm 4. Recall in Chapter 3, a trace must store $\langle x, y, \tau, \xi \rangle$. Like the trace itself, the encoding takes the form of a recursive data structure that stores not only the top level arguments, return values, and scores, but also for every subtrace. The choice map is divided into a hierarchy by callees. We denote the encoding as $\mathcal{D}$ and its dictionary as $D.\lambda$. The dictionary $\lambda$ maps namespace addresses to sub-encodings (and their corresponding arguments, return values, etc.). $\mathcal{D}$ may map to sub-encodings of various trace types, not just DML traces (e.g. combinator traces). Recursive SERIALIZE calls ensure proper handling of subtraces, freeing the top-level from requiring knowledge about how subtraces implement `serialize`. Here we denote the top-level address space as $\tau_A$ for convenience.

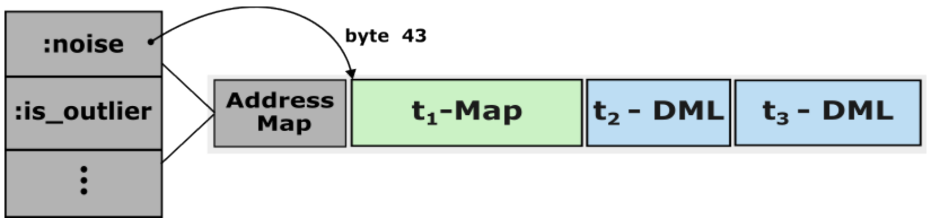


**Figure 4.2: A Dynamic Modeling Trace as a File**. One possible implementation for a DML trace is to organize the subtraces as contiguous chunks. The first block is an "address map" mapping addresses to byte locations. Within each block is a serialized subtrace with possibly smaller blocks of traces. This is analogous to a file system.

The function ENCODE is a third-party serializer (e.g. Serialization.jl for Julia) used to write primitive objects to disk. Separating the logic of SERIALIZE from the choice of ENCODE permits different storage formats and allows users to choose a backend that is most appropriate. Moreover, Algorithm 4 does not specify where each "sub-encoding" is stored physically and leaves the choice to the implementer. Our

[1]It is fine to forgo adding namespaces using 'splicing', but this is not recommended.

implemention for Gen.jl concatentates the subtraces into one file for simple access. As we will explain in Section 4.2, different models and read workloads influence how encodings are organized.

The implementation of DESERIALIZE uses tracing to reconstruct the function identities for the subtraces. Algorithm 5 shows how tracing enables deserialization. The algorithm works by using the stored choices, arguments, and return values of the subtraces to "retrace" the execution of the function. This entails executing the function and intercepting each trace call with known inputs and outputs. This is vital to deal with untraced randomness. By setting the input arguments to predefined values, any untraced randomness that would have affected the arguments is ignored. Similarly returned the stored return values removes the effect of untraced randomness in the return value.

The effect handler for DESERIALIZE has a similar implementation to the other effect handlers [5]. When a user calls DESERIALIZE, a `state` object is created specific for deserialization and exists for duration of the $\mathcal{P}$'s executation. This state contains the trace that is to be built as $\mathcal{P}$ executes - of course the trace built is simply the serialized one. The function $\mathcal{P}$ is executed using the EXEC call that uses the explicit trace call (see Figure 2.2). When a traced call hits address `a`, the handler verifies that $a$ was in the stored $\mathcal{D}.\lambda$. If the address pointed to a value, then the value is decoded added to the reconstructed trace $\mathbf{t}$. Otherwise, the address would have pointed to a subtrace that would need to be deserialized. The subtrace is deserialized using the called generative function `dist`. Using the stored return value `retval` ensures repeatable execution by tracing the addressed values and their corresponding populated values while ignoring the effects of untraced randomness.

**Algorithm 4** Serializing a Trace

```
procedure SERIALIZE($\mathbf{t} = (\mathcal{P}, x, \tau, \omega)$)
    $\mathcal{D} \leftarrow \{\}$
    $\mathcal{D}.\mathrm{x} \leftarrow$ ENCODE($\mathbf{t}.\mathrm{x}$)
    $\mathcal{D}.\mathrm{y} \leftarrow$ ENCODE($\mathbf{t}.\mathrm{y}$)
    $\mathcal{D}.\xi \leftarrow$ ENCODE($\mathbf{t}.\xi$)
    for $a \in \tau_A$ do                                  ▷ Encode top-level choices
        if $\tau[a]$ is value then
            $\mathcal{D}.\lambda[a] \leftarrow$ ENCODE($\tau[a]$)
        else
            $\mathcal{D}.\lambda[a] \leftarrow$ SERIALIZE($\tau[a]$)          ▷ Recursively visit each subtrace
        end if
    end for
    Return $\mathcal{D}$
end procedure
```

**Algorithm 5** Deserializing a DML Trace using Tracing

```
procedure DESERIALIZE-INIT($\mathcal{P}$, $\mathcal{D}$)
    $\mathtt{x}, \mathtt{y}, \xi \leftarrow$ DECODE($\mathcal{D}$.x), DECODE($\mathcal{D}$.y), DECODE($\mathcal{D}.\xi$)
    $\lambda \leftarrow \mathcal{D}.\lambda$
    $\mathbf{t} \leftarrow (\mathcal{P}, x, \{\})$                                    ▷ Build empty trace
    $\mathbf{t}.y \leftarrow y$
    $\mathbf{t}.\xi \leftarrow \xi$
    state $\leftarrow (\mathbf{t}, \lambda)$
    return state
end procedure

procedure DESERIALIZE-HANDLER(state, a, dist)
    $\mathbf{t}, \lambda \leftarrow$ state.$\mathbf{t}$, state.$\lambda$
    if a $\notin \lambda$ then
        error                                             ▷ Address not in stored trace
    end if
    if IS_VALUE($\lambda$[a]) then
        $\mathbf{t}[a] \leftarrow$ DECODE$\lambda[a]$
        $\mathbf{t}[a]$.score $\leftarrow$ DECODE($\lambda[a]$.score)
    else
        $\mathbf{t}' \leftarrow$ DESERIALIZE(dist, $\lambda[a]$)                 ▷ Deserialize subtrace
        $\mathbf{t}[a] \leftarrow \mathbf{t}'$
    end if
    retval $\leftarrow$ state.$\mathbf{t}$.retval                          ▷ Intercept return value
    return retval
end procedure

procedure DESERIALIZE($\mathcal{P}$, $\mathcal{D}$)
    state $\leftarrow$ DESERIALIZE-INIT($\mathcal{D}$)
    retval $\leftarrow \mathcal{P}$.EXEC(state.$\mathbf{t}$.X)         ▷ Begin tracing; retval = $\mathbf{t}$.return
    return $\mathbf{t}$
end procedure
```

### 4.1.1 Optimizing using Lazy Deserialization

While deserialization in the previous section is simple, it is slow because we incur the cost of executing $\mathcal{P}$ to trace the addresses. If any point of the execution is slow, then DESERIALIZE must complete the function before moving onto the next trace call. The work itself to call $\mathcal{P}$ is wasted. Here we describe a simple modification that defers code execution to inference time rather than immediately.

We define a new trace type that is observationally equivalent to DML trace. A trace, **t**', of the new type exposes the same observation interfaces as the normal DML (e.g. GET_CHOICES). When a serialized trace is read back in, there are two options to decode the address and values into **t**'. The first implementation recursively visits the trace and decodes all addresses and values. It is not necessary to rely on tracing as this information is known directly from the dictionary stored in **t**', effectively creating a trace object with all function pointers dropped. On an inference call such as UPDATE, any address that does not have a function pointer assigned to it is assigned one. This avoids paying upfront the entire cost of deserialization and amortizes the cost away to UPDATE or REGENERATE. The second implementation not only avoids executing $\mathcal{P}$ but also defers calling DECODE initially. Instead, calls to DECODE happen at inference type when the address is visited. This is similar to other lazy schemes like copy-on-write.

Figure 4.3 shows an example model that highlights the performance difference between full deserialization and lazily deserializing. The cost for the lazy deserialization is amortized over many calls to UPDATE rather than paying the full cost initially. The table shows the speedup.

```
@gen function model()
  x ~ slow()
  return x
end

@gen function slow()
  sleep(10)
  z ~ bernoulli(0.5)
  return z
end
```

| **Implementations** | **Time per Operation** |
|---|---|
| Serializing Trace | 10.02 s |
| Deserializing Fully | 10.05 s |
| Lazy Initial Cost | 70.53 $\mu$s |
| Updating address `z` | 10.004 s |

**Figure 4.3: Performance of Serializing and Deserializing** The function MODEL runs in roughly 10 seconds to produce a trace. The first value in the table shows the cost of serializing a sample from MODEL. The second measurement is the simpler deserialization implementation. The time is roughly the same as executing the model. The third measurement records the upfront cost of lazily deserialzing which is negligible. The final row shows that when SLOW eventually runs using the lazily deserialized trace, the cost of running the model is incurred once. Timings were measured for Gen.jl.

### 4.1.2 Combinators

The implementations for the combinators are analogous to the ones for DML traces.

**Example 4.1.1** (Map Trace)**.** In Gen.jl, the Map combinator produces *vector* traces consisting of an array subtraces. Section 3.5 suggests one possible implementation: serialize each subtrace and splice the results into one file.

## 4.2 Custom Backends and Batching

So far the discussion has assumed that one trace is serialized at a time. This is fine for a small number of traces, typically used in one-off visualizations or debugging. As the number of serialized traces grows, however, serializing each trace into individual files becomes wasteful. The memory footprint necessary to maintain metadata used in the serialization process grows proportional to the number of samples. Serializing all the samples from a particle filter becomes burdensome if there are numerous large traces. There are computational benefits for (i) selecting a different ENCODE method and (ii) organizing batches of traces to exploit read behavior. For instance, a user may wish to save a collection of particles and latter query for values at a given address. This is typical for visualizations and is characterized by predictable reads into memory. Separated files can suffer from poor locality and do not exploit possible struct of array representations used in many vectorized applications. What format a batch should be formatted in greatly depends on several factors.

*Model Structure* The choice of model affects what addresses are present in a trace. Highly dynamic models produce traces with hetergeonous address structure, and any two trace may have little address overlap. On the other hand, models that conform to simple structure (e.g. linear regression with noise) have predictable traces. "Aligning" traces against their addresses can help reduce the memory footprint of representing the addresses. For example, a model that only produces choice maps of the form $\{a \mapsto x\}$ where $x \in \mathbb{R}$ can be better store a batch by representing the choice maps as a vector of values. Similarly, it may be possible to group argument values or return values into a vectorized representation.

*Read Workloads* Various serialization formats cater to different workloads and store data on disk to optimize for read performance. Workloads may use deserialized traces in different ways. A user who wishes to simply read in values from the choice maps does not need to fully pay the cost of inference and instead can just deserialize the choice map, and a similar scenario occurs when reading query one address for a batch of traces. An application may alternatively wish for cross-language support and require data stored in a universal format rather than a language-specific one.

**Homogenous Models - Arrow Format** For models that produce choice maps with identical or nearly homogenous address structures, it can help to group values by address. The Apache Arrow format organizes data in a columnar representation for flat and hierarchical data, and excels at in-memory column queries.[15] Each column of the Arrow table will correspond to an address in the address universe as shown in Figure 4.4a. If a choice map does not have an address, the corresponding entry in the column is marked with a `missing` value [2]. Figure 4.4c shows the performance of reading one address for a batch compared to deserializing each trace out. Each column resides in contiguous memory and enables for better cache locality.

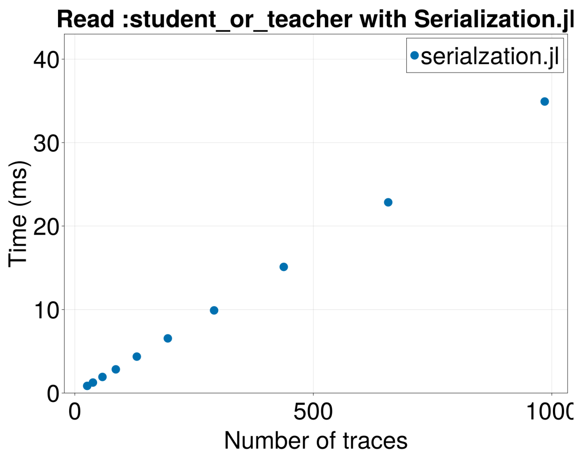
[2] Or some chosen sentinel value.

```
@gen function profession()
  student_or_teacher ~ bernoulli(0.5)
  if student_or_teacher
    salary ~ uniform(0,1)
  else
    grade ~ categorical([0.1,0.5,0.4])
  end
end
```

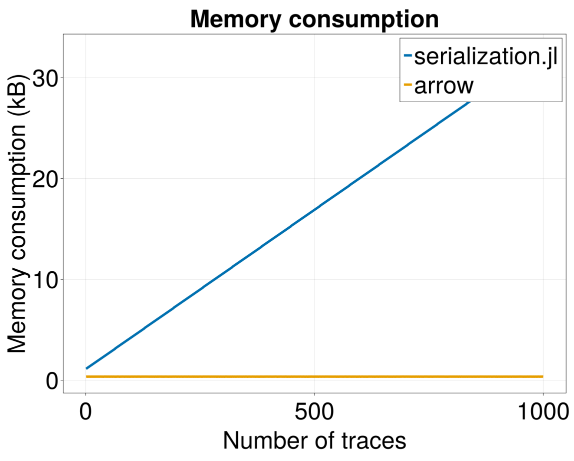
**(a)**

| SAMPLES.ARROW | | |
|---|---|---|
| :SorT | :GRADE | :SALARY |
| 1 | MISSING | 0.8 |
| 1 | MISSING | 0.92 |
| ⋮ | ⋮ | ⋮ |
| 0 | A | MISSING |

**(b)**



**(c)**



**(d)**

**Figure 4.4: Trace file using Arrow** Samples from the model are aggregated into one Arrow file called SAMPLES.ARROW file. Each column groups values by addresses and fills in a MISSING value for any absent addresses. Contiguous layouts enable for better reads. Plot c shows the performance degradation of single file traces using Serialization.jl. If the traces are stored as Arrow table, however, indexing into `:student_or_teacher` for all 1000 traces only took 35.2 *nanoseconds*. While Arrow file and separate trace files when indexing into :STUDENT_OR_TEACHER (:SorT in the table). Plot d shows that Arrow reduces the memory footprint by exploiting struct of array representations.

**Cross-Language Support** As more Gen implementations are developed, cross-language support help clients of different languages interface with each other. Several universal formats exist such as JSON and Protocol Buffers.[14] JSON serves as a versatile and efficient choice for cross-language serialization due to its simplicity and widespread support across programming languages. Its human-readable format makes it easy for developers to comprehend and work with. Furthermore, JSON's key-value pair representation aligns well with the data structures commonly used for addresses such as traces in the Dynamic Modeling Language, allowing for straightforward translation from native data objects to JSON and vice versa. Implementors of serialization can choose one of these formats and use the existing tools to conform the data objects to protocol's specification.

# Chapter 5
# Applications of Serialization in PPLs

In this chapter we show that common use cases of serialization extend to traces. Table 5.1 highlights a collection of workloads that use serialization. We give an overview of each application in the following sections.

| | |
|---|---|
| Section 5.1 | Storage and reproducibility |
| Section 5.2 | Checkpointing and external memory computation |
| Section 5.3 | Remote procedure calls |
| Section 5.4 | Amortized Inference |

**Table 5.1:** Examples of use cases for serialization in data analysis and computing.

## 5.1 Storage and Reporducibility

The choice map in a trace is informative to understand the generative process that produced that trace and can be accessed using the `choices` interface. A user could inspect the choice map for a wide variety of applications that may or may not be related to inference itself.

**Visualizations**

It is common to visualize samples as empirical distributions to analyze model behavior in experiments. Serializing out traces is especially useful if the gnerative model is slow to sample from so that the user does not need to run the model each time to generate visuals. Moreover, it is possible that the user runs one-off models that collect data once and must be saved onto disk for long-term storage.

**Reproducibility**

Saved out traces can be used to replicate experiments by using an initial trace as a seed for inference. Serialization helps ensure that the trace data remains unchanged, enabling reproducibility of the exact conditions under which the generative model ran. Figure 5.1 shows an example.

**Versioning**

Serialization also enables users to version out data and compare across different experimental methods. Users can incrementally test iterations of an experiment with the same underlying data, simplifying the work needed to make comparison or ablation studies. For instance, serialization is useful in diagnosing the convergence of inference strategies also shown in Figure 5.1. Rather than starting at the beginning of inference, the user can begin each algorithm at any point of interest to explore differences.

**Data Accessability**
Serialized data can be shared with other users, facilitating collaboration.

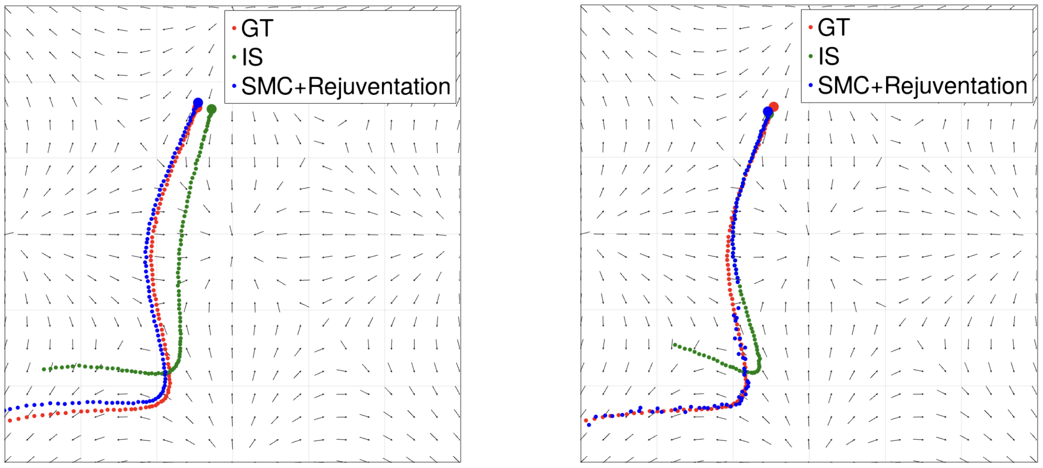


**Figure 5.1: Experiement Reproducibility** Serialization enables comparing traces representing the estimated trajectories from different inference algorithms. The left panel shows a particle's ground truth trajectory in red subject to a field. Two trajectories produced by importance sampling and SMC respectively are also plotted. The SMC algorithm uses a data-driven proposal which matches the trajectory better. The right panel shows the same algorithms agree for a portion of the trajectory using an initial trace. The quality difference becomes clearer after a sudden change in the trajectory and the blue trajectory self-corrects.

## 5.2 Checkpointing Progress and Out-of-Core Computation

In common inference algorithms, multiple iterations are carried out to achieve convergence (e.g., MCMC) or process sequential data (e.g., time series). Checkpointing enables the saving a single or groups of traces to a stable medium, mitigating the potential for data loss in the event of a crash during the inference algorithm's execution. This method eliminates the necessity to restart the entire computation. By serializing traces to disk at regular intervals, programs that error unexpectedly can resume computation by reading the latest saved traces.

A similar application is to use serialization to manage collections that exceed the available memory. Particle filters rely on having a sufficient number of particles for good convergence. In most settings, all the particles can fit in virtual memory without reaching the memory limit. However, it is possible for a model to allocate a large chunk of memory to produce traces. By only keeping a handful of particles and relegating the rest to disk, a particle filter can operate with low memory resources. A naive scheme would be to save out all the traces to disk and read each individually when during calls to UPDATE. Of course, this increases latency so more sophisticated scheduling can improve performance and can leverage ideas from database managers, as these systems typically save pages to disk and use buffer pools to manage the pages currently in memory. The pseudocode below shows an SMC step with one particle active at a time. Each particle is read back into memory, updated to a new trace, and serialized back to disk. The resampling step selects a new particle collection and deletes unused particles.

```
for i=1:N
  trace = deserialize_trace(i)
  trace, w = update(gen_fn, trace, obs)
  weight[i] = w
  serialize_trace(trace)
end
new_traces, discard_traces = resample(weights)
flush(discard_traces)
```

An interesting application of checkpointing is for particle rejuvenation-like sampling. Particle rejuvenation is necessary to widen the support of the particle collection and is useful to prevent collapses. Checkpoints can be used to save particles of particular interest during filtering and later read back into memory to as resampled particles for rejuvenation. Algorithm 6 illustrates the high level procedure. There, $R$ is the rejuvenation function which also takes the checkpoint traces and produces a new set of particles.

**Algorithm 6** Particle Rejuvenation with Checkpoints

```
Input: Particles $\{\mathbf{x}^{(i)}\}_{i=1}^{N}$, Rejuvenation function $R$, Saved particles $S = \{x_s^{(i)}\}_{j=1}^{M}$
new_particles ← $\emptyset$
new_particles$^{(i)}$ ← $R(\mathbf{x}^{(i)}, S)$ for $i \in [n]$        ▷ Rejuvenate particle $i$
return new_particles
```

## 5.3 Remote and Distributed Computing

For more computationally expensive models or inference, parallelization and remote calls can help speed up performance. To facilitate communication, programs can serialize traces and transmit data the across the network. Section 4.2 discussed factors that can influence how traces are formatted for efficient storage, but here we explain when to call SERIALIZE to achieve basic process communication. Transmitting a trace occurs in one of two places. The first is across separate calls to the interface methods such as consecutively calling UPDATE on a particle filter. The second is when executing a sub-generative models.

**Parallelism Over Particles** A particle filter can run in parallel by dividing particles over separate process. Algorithms like SMC may require processing particles after each step (e.g resampling), and so traces must be aggregated. Worker processes can serialize traces onto a stream or channel and transmit the traces a master process for aggregation. Figure 5.3 shows pseudocode for one step of a particle filter that uses a master node to resample the particle collection. The left snippet shows the worker code which updates each trace allocated to it and serializes back the results to the master. The right snippet shows the the master process aggregating traces and then (assuming the all workers finished) performs resampling.

```
while is_empty(jobs)
  (trace, args) = take(jobs)
  trace = deserialize(gen_fn, trace)
  new_trace =
    update(gen_fn, trace, args...)
  new_trace = serialize(new_trace)
  push(aggregate, data)
end
```

```
# Allocate jobs
for i=1:N
  push(jobs, serialize(traces[i]))
end
...
# Wait for workers
# Deserialize returned traces
new_traces = ...
new_traces = resample(new_traces)
```

**Figure 5.3**

### Remote Procedure Calls

Serialization can also help to invoke remote function on a different process. We consider a client and server where the client does not have access to the function definition use generate a trace. The previous implementation for parallelized particles extends here as well if the client wishes to make a call to the remote generative function and obtain the trace. For example, the client can send arguments to the server for a SIMULATE call, and the server returns back a serialized copy of the trace. The copy can be downloaded and inspected for its contents like the choices, and sent back to the server for future inference calls.

It is also possible to make remote calls *inside* a generative function, but these calls are not traced by default in Gen.jl as they are treated as black-boxes. We demonstrate how to use a custom DSL to construct remote generative functions that *can* be traced for the DML. In Appendix A, we show how to extend the tracing implementation of Gen.jl to add a new macro for remote traced calls when functions are globally defined across processes, but this does not require serialization.

Here we assume a tracing implemenations of generative functions and specifically focus on Gen.jl, but a similar approach works for non-traced modeling languages. The same principles apply but now a custom generative function handles sending and saving traces from the server during tracing without the user handling this between calls. Gen.jl supports writing custom generative functions so long as they satisfy the Gen interfaces. Now the custom generative function implements the Gen interfaces using effect handlers similar to the ones in the Dynamic Modeling language. Algorithm 7 shows the custom effect handler for `simulate` on the client machine. `dist` is assumed to be the custom generative function [1]. The SIMULATE call makes a remote call to the server using the provided arguments and retrieves back the serialized contents. The handler assigns to the trace a reference (e.g. file path for traces on disk) for the subtrace at the intercepted address. Algorithm 8 shows the entry point to the server's SIMULATE. By abstracting away the subtrace for the remote machine and storing it to disk without ephemeral data objects, the client is able to communicate with machines that do not share common functions. The other methods can be implemented similarly. Extending the other interfaces is similar except that old trace must be deserialized when the server receives it.

**Algorithm 7** Remote Simulate on Client

```
procedure SIMULATE-HANDLER(state, $x$, dist)
    retval, score, trace_handle $\leftarrow$ RPC_SIMULATE_P($x$)
    state.$\tau[a]$ $\leftarrow$ trace_handle
    state.$\tau[a]$.retval $\leftarrow$ retval
    state.$\tau[a]$.score $\leftarrow$ score
    Return retval
end procedure
```

**Algorithm 8** Server SIMULATE RPC handler

```
procedure RPC_SIMULATE_P($x$)
    t $\leftarrow$ $\mathcal{P}$.SIMULATE($x$)
    file_handler $\leftarrow$ t.SERIALIZE()
    Return t.retval, t.score, file_handler
end procedure
```

[1]Julia supports multiple dispatch and can therefore dispatch on function type.

## 5.4 Training Approximate Distributions

This section discusses EUBO optimization using several case studies and explores scenarios where serialization is helpful. The first part deals with approximating energy-based models where direct sampling is often expensive and requires MCMC methods. We show that by serializing samples from the expensive energy-models, we can speed up training without a significant hit in performance. Likewise, the second section explores training neural proposals for pose inference probabilistic programs. Pose inference via inverse graphics can strain the renderer for high resolution images, so serialization can offer speed-up advantages. The final section examines training neural surrogates for importance sampling. A neural surrogate approximates the score of the target distribution and has a virtually identitical training process. Although this chapter focuses on EUBO optimization, the same ideas discussed here can be extended to other objectives (e.g. SDOS). [6]

### 5.4.1 Amortized Inference for Expensive Generative Models

Recall the forward KL objective in Section 2.3 is defined as $D(p(z|x)||Q_\theta(z;x))$ for target posterior $p(z|x)$ and approximate distribution $Q_\theta(z;x)$. Minimizing the divergence corresponds to learning parameters $\theta$ for $Q_\theta(z;x)$ which in this chapter is a parameterized SIR. Let $Q_s$ be the scorer and $Q_p$ be the proposal for SIR. It is the choice of the model implementer to parameterize $Q_s$ or $Q_p$. Samples from the generative process $p(x,z)$ are fed into the importance sampler as training data. Gradients are computed as

$$-\int p(x,z)\nabla_\theta \log Q_\theta(z;x) \tag{5.1}$$

$$\tag{5.2}$$

While it is common to assume that it is easy to sample from $p(x,z)$ using the observationally equivalent factorization $p(z)p(x|z)$, this assumption is not always true. For example, if $p(x|z)$ is an energy function or is itself a posterior, then sampling according to $p(x|z)$ requires approximation techniques such as MCMC. Figure 5.4 shows samples from example distributions whose generative processes are expensive no matter the factorization. The left panel shows samples $(x,y,r^2)$ using MH steps conditioned on $r$ using an unnormalized energy function. The remaining panels shows sample lattices from the Ising model using an efficient Gibbs block sampler. [2] Both distributions require many iterations to produce samples according to their respective energy functions.

[2]Recall that a lattice $x \in \{\pm 1\}^N$ from the Ising model with no external field has energy $E(x;\beta) = \exp(-\beta H(x))$ and $H(x) = \sum \theta_{ij} x_i x_j$. The value $\beta$ is the inverse temperature and influences the distribution of $\pm 1$ on the lattice.

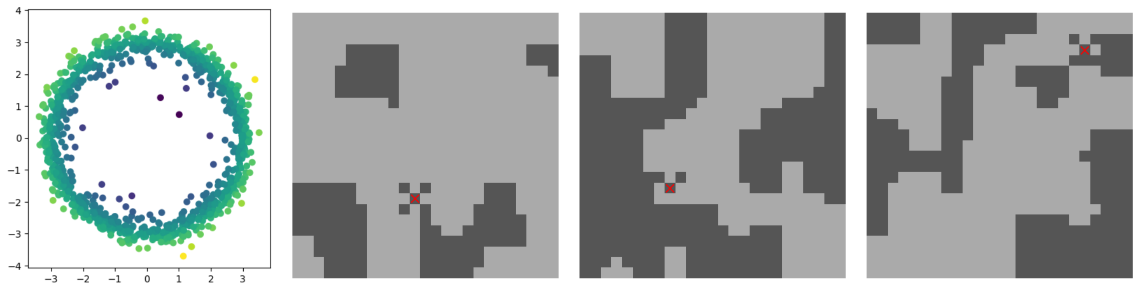


**Figure 5.4:** Sampling from from energy functions requires iterating to get good approximations. Often this cost cannot be avoided. The first plot corresponds to the energy function $E(x, y, r^2) = \exp(-2(x^2 + y^2 - r^2))$. The remaining figures are Ising lattices with slightly different interaction coefficients at $\beta = 0.5$. The red cross indicates the impurity of the lattice which induces a checker-like pattern.

```
procedure CIRCLE-MODEL(N)
    $x \sim$ UNIFORM
    $y \sim$ UNIFORM
    $r^2 \sim$ UNIFORM(0,1)
    for i=1:N do
        $x, y, r^2 \sim \mathrm{MH}(x, y, r^2)$
    end for
end procedure
```

```
procedure DIRTY-ISING(N)
    $\alpha \sim$ DIRTY-PRIOR()
    grid$\sim$ GRID-PRIOR()
    for $i = 1 : N$ do                ▷ Gibbs steps
        grid$\sim$ GIBBS(grid, 1)
        grid$\sim$ GIBBS(grid, 2)
    end for
    Return grid
end procedure
```

**Figure 5.5**

**Regression Proposal** The left panel of Figure 5.5 shows the generative process for the samples that lie on concentric circles, and the importance sampler distributions $(Q_s, Q_p)$. The distribution $p(x, y, r^2)$ is proportional to the energy function $E(x, y, r^2) = \exp(-\beta H(x, y, r^2))$ where $H(x, y, r^2) = x^2 + y^2 - r^2$, and the distribution of interest is $p(r^2|x, y)$. Since $p(r^2|x, y) \propto p(x, y, r^2)$, we can use the generative process as the unnormalized target distribution for importance sampling. This means that the scorer $Q_s(r^2; x, y) \propto E(x, y, r^2)$ is the same energy function and is cheap to compute - it is responsible for *scoring* but not *sampling*. The neural proposal $Q_p$ shown in left panel of Figure 5.6 uses regression coefficients to predict the mean of a Gaussian for $r^2$. The optimal coefficients are clearly $a = 1$, $b = 1$, $c = 0$, $d = 0$, and $e = 0$.

**Convolutional Network Proposal** The right panel of Figure 5.5 descibes a modified Ising model that has a "impure" magnetic interaction. In a "pure" lattice, $\theta_{ij} = -1$ for all $i, j$. Instead, the latent $\alpha$ determines where a small block of $\theta_{ij} = 1$ is centered. Figure 5.4 shows how the impure interaction influences the sample lattice. The target distribution is $p(\alpha|x)$ where $x$ is a lattice. The proposal $Q_p(\alpha; x)$ maps lattice images to values $\alpha$ for the mean of a Gaussian. The target distribution is $p(\alpha|x)$ where $x$ is a lattice.

```
procedure CIRCLE-PROPOSAL(X,Y)
    Parameters: $(a, b, c, d, e)$
    $\mu = ax^2 + by^2 + cy + dx + e$
    $r^2 \sim$ NORMAL($\mu, \sigma$)
end procedure
```

```
procedure ISING-PROPOSAL(GRID)
    Parameters: $\theta$
    $\mu_\alpha =$ CONVNET$_\theta$(GRID)
    $\alpha \sim$ NORMAL($\mu_\alpha, \sigma$)
end procedure
```

**Figure 5.6**

**Convolution Network for Pose Inference** Here we consider a network that is not energy based but requires substantial deterministic computation. Recent work has used probabilistic programs to infer latent object poses by using rendering [8]. Inverse graphics uses latent object models and poses to render out scenes and scores hypothesis by how well they match the observed images. Although the original work defined an extensive model for multi-object scenes, we restrict to the case of one object.

A pose is an element of SE(3) and is composed of a rotation and a translation. The particle filter for inference tracks moving objects by predicting the latent pose at each time step using an enumerative proposal. By gridding the latent space near the last known estimated pose, the proposal renders out each hypothesis as an depth map. Figure 5.7 shows a diverse set of hypothesis of an object given a known location in space. The depth map is scored against the observed depth map using the point cloud likelihood defined in [8]. The particle filter resamples particles with high importance weight. While enumeration is robust, full enumeration over a large latent space can be expensive as each particle needs to be rendered out.

Rendering is GPU accelerated, but the enumeration cost still remains. A neural proposal can help enumeration by suggesting small grid sizes around where the object is likely to be. We opted for a convolution network that takes a depth map and predicts the translation component of the pose. The predicted translation is then perturbed by Gaussian noise. The bottom row of Figure 5.7 shows the diversification of the proposed samples with the trained neural proposal.

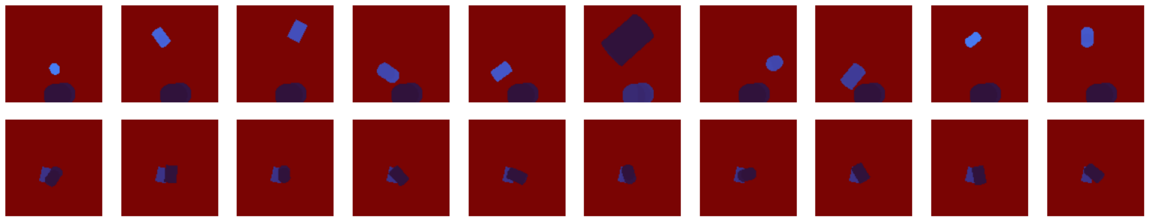


**Figure 5.7: Proposals for Object Pose** On the top row, the ground truth cylinder lies on the bottom of the image. Each image shows the proposed pose of an uninformed prior. On the bottom row, the trained network concentrates the proposals near the true translation component.

### 5.4.2 Training

Slow generative models create a computational bottleneck when training as the gradient in Equation 5.2 requires sampling from $p(x, z)$. Using fresh samples provides large data coverage but is wasteful as samples in one gradient step are discarded for the next step. In conventional machine learning scenarios, training typically involves drawing samples from a static dataset, and batches might reuse samples throughout the training procedure. The same principles apply here where we serialize out samples to create datasets. The generative processes can be sampled from indefinitely, so there is flexibility on how often samples are saved out and reused for future gradients. We compute gradients using several training strategies that save out traces at different frequencies. Here, we list four strategies but of course different combinatons are possible.

- **Fresh Samples -** Gradients are computed using new samples from $p(x, z)$, and are immediately discarded. If a gradient estimator uses $K$ samples and evaluates a gradient $N$ times, then $KN$ samples are needed.
- **Static Dataset -** Samples are generated ahead of time and aggregated into one dataset.
- **Mixed Dataset -** Initially a static dataset is constructed and periodically regenerated.
- **Regenerated Dataset -** For generative functions that use MCMC, samples do not need to be generated from scratch and can instead be sampled using previous observations. In addition to generating samples from scratch, models that use MCMC for sampling can use old samples and regenerate by running the chain more.

# Chapter 6
# Conclusion

We conclude by assessing the serialization design and highlight other potential applications for serialization. This thesis has demonstrated that it is possible to serialize in Gen's implementations while guaranteeing observational equivalence for later inference. Traces read from disk remain valid even if generative function identies are detached and remain compositional with the other generative function interfaces. Moreover, the interfaces provide several degrees of freedom in specifying a trace's format on disk. Depending on several factors such as generative function structure, choice data types, and vectorization, the user can customize the serialization format for better read performance. A key feature of these interfaces is that that they remain composable and consistent with Gen's other interfaces even with hierarchical models, stochastic branching, and even certain cases of untraced randomness.

## 6.1 Future Work

The serialization implementations discussed in Section 4 primarily covered the Gen's Dynamic Modeling Language but left open the possibility for the Static Modeling Language. Greater care is needed because compilation of static generative models may change the generative functions identity, its AST structure, or even the AST tokens themselves and thus the same procedure for the DML would not work. For example, Julia's implementation of the SML constructs an entirely new function whose identity and variables are entirely different from the source - it is possible that restarting the Julia environment renders these objects invalid. As of this thesis, Gen's SML compiler applies static analysis to infer model structure and this could be leveraged for deserialization. Supporting the static language brings up a broader point about how compiler design influences which internal internal data structures can and cannot be serialized. More sophisticated compilers could hold more auxilliary data in the trace to benefit inference, and serialization implementations would need to cover these cases.

Moreover, there is limited support for serializing traces from models that sample *functions* as choices. For example, [16] proposes a generative model that fits time series data against stochastic generative models also written in Gen, and thus serializing would require saving out functions as *values*. Presumably, these sampled functions have a lowered represention (e.g. an AST) composed of simple types that *can* be serialized. For these cases, SERIALIZE could be specialized to write the lowered representation instead.

It is necessary to consider the security of the serialization implementation for security-sensitive settings. Similar to Python's `pickle`, DESERIALIZE reads in arbitrary pointers from memory and does not check what is deserialized. It is conceivable that a malicious user may be able to execute arbitrary code upon deserialization although the current design softly mitigates this so long as a user has only trusted trace types in the namespace.

The discussion in Section 5.4 trained variational importance samplers using saved out datasets. Future experiements could quatify how well neural proposals using saved out static datasets compare to learned proposals that continuously sample from the generative model.

# Appendix A
# Remote Generative Function Calls

The benefit of using a tracing system in Gen is that it enables for large code reuse for other types of tracing. Figure A.1 shows the use of two new (desugared) tracing macros, GEN_RPC and GEN_FETCH. In a normal distributed setting, clients submit RPCs to a remote machine and receive back a *future*. This future does not immediately hold the return values of the function but serves as a token the client can query. Upon a *fetch*, the client blocks and only resumes execution after receiving the return value. This is identical in the case of generative functions in Gen.jl. The model first calls a worker process using GEN_RPC with a process id and receives back a future. The model then makes a call to GEN_FETCH, blocking the execution until the worker process returns the subtrace.

```
@gen function really_slow_sum(args)
    futures = []
    for i=1:10
        future = gen_rpc(i % nproc, state, remote_gen_fn, i,  (args,))
        push!(futures, future)
    end
    ... # Work

    results = []
    for i=1:10
        push!(results, gen_fetch(state))
    end
    final = sum(results)
    return final
end
```

**Figure A.1: Remote Tracing Macros** The left snippet shows an example using the GEN_CALL and GEN_FETCH using the expanded out trace calls. Each invocation of GEN_CALL sends work to an available process. The model can continue working while the remote processes finish their respective tasks, and the results are aggregated at the end.

The same rules for common distributed computing apply here for generative models. For example, using the unfetched future may produce undefined behavior. Moreover, in our implementation calling GEN_FPC does not immediately append to the execution trace and instead defers this to the corresponding call to GEN_FETCH. For example, if during the time between making the RPC and fetching future (e.g. during `Work`) one of these addresses is populated, then the corresponding fetch call will fail.

# Bibliography

[1] David M. Blei, Alp Kucukelbir, and Jon D. McAuliffe. Variational inference: A review for statisticians. *ArXiv*, abs/1601.00670, 2016.

[2] Yuri Burda, Roger Baker Grosse, and Ruslan Salakhutdinov. Importance weighted autoencoders. *CoRR*, abs/1509.00519, 2015.

[3] Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of Statistical Software*, 76(1):1–32, 2017.

[4] N. Chopin and O. Papaspiliopoulos. *An Introduction to Sequential Monte Carlo*. Springer Series in Statistics. Springer International Publishing, 2020.

[5] Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. Gen: A general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 221–236, New York, NY, USA, 2019. ACM.

[6] Justin Domke. An easy to interpret diagnostic for approximate inference: Symmetric divergence over simulations. *ArXiv*, abs/2103.01030, 2021.

[7] Hong Ge, Kai Xu, and Zoubin Ghahramani. Turing: A language for flexible probabilistic inference. In Amos Storkey and Fernando Perez-Cruz, editors, *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics*, volume 84 of *Proceedings of Machine Learning Research*, pages 1682–1690. PMLR, 09–11 Apr 2018.

[8] Nishad Gothoskar, Marco Cusumano-Towner, Ben Zinberg, Matin Ghavamizadeh, Falk Pollok, Austin Garrett, Josh Tenenbaum, Dan Gutfreund, and Vikash Mansinghka. 3dp3: 3d scene perception via probabilistic programming. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 9600–9612. Curran Associates, Inc., 2021.

[9] Ghassen Jerfel, Serena Wang, Clara Wong-Fannjiang, Katherine A. Heller, Yian Ma, and Michael I. Jordan. Variational refinement for importance sampling using the forward kullback-leibler divergence. In Cassio de Campos and Marloes H. Maathuis, editors, *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence*, volume 161 of *Proceedings of Machine Learning Research*, pages 1819–1829. PMLR, 27–30 Jul 2021.

[10] Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. *CoRR*, abs/1312.6114, 2013.

[11] Tuan Anh Le, Atilim Gunes Baydin, and Frank Wood. Inference Compilation and Universal Probabilistic Programming. In Aarti Singh and Jerry Zhu, editors, *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, volume 54 of *Proceedings of Machine Learning Research*, pages 1338–1348. PMLR, 20–22 Apr 2017.

[12] Alexander K. Lew, Marco Cusumano-Towner, and Vikash K. Mansinghka. Recursive Monte Carlo and variational inference with auxiliary variables. In James Cussens and Kun Zhang, editors, *Proceedings of the Thirty-Eighth Conference on Uncertainty in Artificial Intelligence*, volume 180 of *Proceedings of Machine Learning Research*, pages 1096–1106. PMLR, 01–05 Aug 2022.

[13] Brooks Paige and Frank D. Wood. Inference networks for sequential monte carlo in graphical models. In *International Conference on Machine Learning*, 2016.

[14] Felipe Pezoa, Juan L Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. Foundations of json schema. In *Proceedings of the 25th International Conference on World Wide Web*, pages 263–273. International World Wide Web Conferences Steering Committee, 2016.

[15] Neal Richardson, Ian Cook, Nic Crane, Dewey Dunnington, Romain François, Jonathan Keane, Dragoș Moldovan-Grünfeld, Jeroen Ooms, and Apache Arrow. *arrow: Integration to 'Apache' 'Arrow'*, 2023. https://github.com/apache/arrow/, https://arrow.apache.org/docs/r/.

[16] Feras A. Saad, Brian J. Patton, Matthew D. Hoffmann, Rif A. Saurous, and V. K. Mansinghka. Sequential Monte Carlo learning for time series structure discovery. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 29473–29489. PMLR, 2023.

[17] Surya T. Tokdar and Robert E. Kass. Importance sampling: a review. *WIREs Computational Statistics*, 2(1):54–60, 2010.

[18] Tongzhou Wang, YI WU, Dave Moore, and Stuart J Russell. Meta-learning mcmc proposals. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[19] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992.